

TAME: A PVS Interface to Simplify Proofs for Automata Models *

Presented at UITP 98, Eindhoven, Netherlands, July 13-15, 1998

Myla Archer, Constance Heitmeyer, and Steve Sims
Code 5546, Naval Research Laboratory, Washington, DC 20375
{archer, heitmeyer, sims}@itd.nrl.navy.mil

Abstract

Although a number of mechanical provers have been introduced and applied widely by academic researchers, these provers are rarely used in the practical development of software. For mechanical provers to be used more widely in practice, two major barriers must be overcome. First, the languages provided by the mechanical provers for expressing the required system behavior must be more natural for software developers. Second, the reasoning steps supported by mechanical provers are usually at too low and detailed a level and therefore discourage use of the prover. To help remove these barriers, we are developing a system called TAME, a high-level user interface to PVS for specifying and proving properties of automata models. TAME provides both a standard specification format for automata models and numerous high-level proof steps appropriate for reasoning about automata models. In previous work, we have shown how TAME can be useful in proving properties about systems described as Lynch-Vaandrager Timed Automata models. TAME has the potential to be used as a PVS interface for other specification methods that are specialized to define automata models. This paper first describes recent improvements to TAME, and then presents our initial results in using TAME to provide theorem proving support for the SCR (Software Cost Reduction) requirements method, a method with a wide range of other mechanized support.

1 Introduction

A good theorem prover interface has several goals. At the first, most immediate level, an interface should be designed to make the customary interactions with the theorem prover as convenient as possible. For example, the interface should make it easy and convenient to display known facts (such as theories and lemmas) [19] and should provide automated support for selecting applicable proof rules [21, 20]. It should also facilitate the selection of proof rules, lemmas to apply, expressions inside a proof goal to be used as instantiations, and so on. At a second, somewhat deeper, level, an interface should provide supplementary services in the theorem prover itself. For example, a user should be able to add annotations to proofs [13] or to obtain human-understandable proof scripts [5]. At a third, even deeper, level, an interface should provide derived proof rules that allow the user to reason in familiar ways, e.g., using his favorite logic and syntax. Isabelle [15, 14] is designed particularly to support such interfaces, but any “programmable” prover can support them. For example, a proof assistant for the duration calculus has been built on top of PVS [18].

We are developing a tool called TAME (Timed Automata Modeling Environment) [1, 3, 2] that provides interface features at both the second and third levels for PVS (Prototype Verification System) [17]. In particular, TAME supports reasoning about automata models by providing specialized PVS proof steps that are appropriate for proving properties of such models and that automatically annotate both proof goals and saved proofs with meaningful labels and comments. Currently, TAME is comprised of these specialized proof steps together with a set of standard theories and automata templates upon which the steps rely, and in itself has no interface features at the first level. However, we have recently investigated the integration of TAME into a set of tools called the SCR toolset [9, 6, 7]. The SCR tools are designed to support editing and performing various kinds of analysis on requirements specifications of control system software. Once TAME has been fully integrated into the SCR toolset, the user who wishes to apply TAME to an SCR specification will have first-level interface support, including the extensive interface support already provided by the SCR tools. Ultimately, we also plan to provide direct first-level interface support for TAME.

Our goals in developing TAME are somewhat different from the goals of the developers of the duration calculus assistant. Rather than supporting a particular logic, TAME supports proof steps “natural” to humans reasoning about automata models [2]. To the extent feasible, TAME hides the raw PVS logic from the user who is proving properties of automata, since interacting with PVS in its “raw” form has its difficulties. For example:

- Formula numbers may be needed, e.g., in skolemization or instantiation; this can lead to non-portability of proofs or difficulty in defining generic strategies.

*This work is funded by the Office of Naval Research. URLs for the authors are
<http://www.itd.nrl.navy.mil/ITD/5540/personnel/{archer,heimeyer}.html>

- Rather obscure proof steps, e.g., the APPLY-EXTENSIONALITY rule, are occasionally needed to perform reasoning that is “obvious” to a human.
- Quantified formulae may need as instantiations complex, sometimes multi-line, expressions that are cumbersome to supply using the standard PVS interface.
- Proofs are often structured more for the needs of the prover than for the needs of the human constructing the proof. For example, splitting an hypothesis can produce several proof goals whose significance is hard to understand, and require the user to provide obscure proof steps. This is particularly a problem when, as often happens, the split hypothesis is an implication and the user’s arrival at the later proof goal or goals is delayed in time.
- Many proof steps provided by raw PVS are either too small or too large when compared with reasoning steps natural to a human. Proofs that require many small steps become quite tedious. When a large step involving the PVS decision procedures is used in a proof, the significance of resulting subgoals (if the step does not complete the proof) is obscure.

The problems listed above are not for the most part peculiar to PVS. They or their analogues are likely to arise with any mechanical theorem prover. However, these difficulties demonstrate why following a more natural style of reasoning in PVS requires improvements to the PVS interface.

As indicated in [2], we have successfully defined and applied TAME proof steps which are more “natural” for humans than those provided directly by PVS. An important reason for our success is that TAME is an interface specialized for particular proof styles and particular models. However, certain barriers prevented us from doing more. Recent enhancements to PVS, soon to be incorporated in the general PVS release [16], remove some of these barriers and allow more sophisticated proof steps that avoid the difficulties listed above. The enhancements allow us to provide more information to the user about the proof goals during the course of a proof and to automatically provide better documentation in PVS proof scripts.

We believe that specialized interfaces such as TAME will encourage the more widespread use of theorem provers in practical software development. Our experience in developing TAME suggests that adding certain capabilities to existing general theorem proving systems can encourage the development of specialized interfaces for these provers.

The remainder of this paper is organized as follows. Section 2 reviews PVS, timed and non-timed automata, TAME, and SCR. Section 3 describes in detail the improvements to TAME made possible by the PVS enhancements. Section 4 provides examples that illustrate the new features of TAME. Section 5 discusses how TAME has been customized to support SCR automata models, and the progress we have made in integrating TAME into the SCR toolset. Finally, Section 6 discusses some issues that arise in developing specialized interfaces for PVS and other provers and our future plans for TAME.

2 Background

2.1 PVS

PVS [17] is a higher order logic specification and verification environment developed by SRI. Proof steps in PVS are either *primitive* steps or *strategies* defined using primitive proof steps, applicative Lisp code, and other strategies. Strategies may be built-in or user-defined. PVS’s support for user-defined strategies makes it possible to implement specialized prover interfaces such as TAME on top of PVS. Recently, several enhancements to PVS have been developed at SRI. These include new features better supporting specialized interfaces. The new features include support for labeling formulae appearing in proof goals, support for documenting proof structure and proof steps (both interactively and in proof scripts) through comments, and the availability of certain fine-grained proof steps. The addition of some new access functions and documentation allows computations based on the internal data structures maintained by PVS. These enhancements, which will ultimately become standard features of PVS, have led to major improvements in TAME.

2.2 LV Timed Automata and IO Automata

An LV timed automaton is a very general automaton, i.e., a labeled transition system that incorporates the notions of current time and timed transitions. An automaton need not be finite-state: for example, the state can contain real-valued information, such as the current time, the water level in a boiler, velocity and acceleration of a train, and so on. LV timed automata can have nondeterministic transitions; this is particularly useful for describing how real-world quantities change as time passes, given upper bounds on their rate of change.

The following definition of timed automaton, based on the definitions in [8], was used in our case study of a deterministic timed automaton [1]. A *timed automaton* A consists of five components: (1) $states(A)$, a (finite or infinite) set of states, (2) $start(A) \subseteq states(A)$, a nonempty (finite or infinite) set of initial states, (3) a mapping now from $states(A)$ to $R^{\geq 0}$, the non-negative real numbers, (4) $acts(A)$, a set of actions (or events),

which include special *time-passage* actions $\nu(\Delta t)$, where Δt is a positive real number, and *non-time-passage* actions, and (5) $steps(A) : states(A) \times acts(A) \rightarrow states(A)$, a partial function that defines the possible steps (i.e., transitions). This definition describes a special case of LV timed automata that requires the next-state relation, $steps(A)$, to be a function. Careful use of the Hilbert choice operator ϵ , allows us to use the same basic definition in the nondeterministic case as well [3]. The definition of IO automata is similar, except that it has no references to time.

Actions (or events) may at any point be enabled or disabled. The typical specification of an LV timed automaton or an IO automaton describes the actions in terms of preconditions under which they are enabled, and effects on the state. Below, we refer to these preconditions as *specific* preconditions; other, uniformly applied components of the full precondition may also exist, such as general timing constraints in a timed automaton. The *transitions* (or steps) of an automaton correspond to the state changes induced by enabled actions. The *reachable states* of an automaton are those states that can be reached from an initial state via a sequence of zero or more transitions.

The properties of automata that one wants to prove fall into three classes: (1) state invariants, i.e., properties of all reachable states, which are typically proved by induction; (2) simulation relations; and (3) ad hoc properties of certain execution sequences. Proofs in both (1) and (2) have a standard structure with a base case involving initial states and a case for each possible action. They are thus especially good targets for mechanization. The proof examples in this paper all fall into class (1).

2.3 TAME

TAME provides a standard template for specifying automata, a set of standard theories, and a set of standard PVS strategies. The TAME template, originally intended for specifying LV timed automata, provides a standard organization for defining an automaton. To define either a timed or non-timed automaton, the user supplies the following six components: (1) declarations of the non-time actions, (2) a type for the “basic state” (usually a record type) representing the state variables, (3) any arbitrary state predicate that restricts the set of states (the default is **true**), (4) the preconditions for all transitions, (5) the effects of all transitions, and (6) the set of initial states. The user may optionally supply declarations of important constants, an axiom listing any relations assumed among the constants, and any additional declarations or axioms desired.

To support mechanical reasoning about automata using proof steps that mimic human proof steps, we have constructed a set of standard strategies using PVS, and included these as part of TAME. These strategies are based on the set of standard theories and certain template conventions. For example, the induction strategy, which is used to prove state invariants, is based on a standard automaton theory called **machine**. To reason about the arithmetic of time, we have developed a special theory called **time_thy** and an associated simplification strategy called **TIME_ETC_SIMP** for time values that are either non-negative real values or ∞ . The important template conventions include a standard naming scheme and a standard format for lemmas of certain classes, such as state invariant lemmas. A more detailed description of TAME in its original form is available in [1]. Some enhancements to TAME in support of the verification of hybrid automata are described in [3].

Using the recent enhancements to PVS described in Section 2.1, we have simplified the original set of TAME strategies and provided several new strategies. In addition, progress has been made towards making both proof scripts and interactive proof goals “literate”. Section 3 describes the details of these improvements to TAME, and Section 4 provides examples of their use.

Although TAME was originally developed for reasoning about LV timed automata, it is equally useful for non-timed IO automata [4] and easily adapted to many other automaton models. We have begun to apply TAME to SCR automata (see Section 2.4). For this new application, a slight modification to the TAME template has proved important for proof efficiency. Additional strategies that complete many state invariant proofs totally automatically seem possible. Section 5 presents the results of our initial experiments.

2.4 The SCR Requirements Method and Toolset

The SCR (Software Cost Reduction) requirements method is a formal method for specifying and analyzing the requirements of safety-critical control systems. Since its introduction in 1980 [10], SCR has been applied successfully to a wide range of critical systems, including avionics systems, space systems, telephone networks, and control systems for nuclear power plants. A set of software tools, called SCR* [9, 6, 7], has been constructed to support the SCR method. In addition to a *specification editor* for creating a specification and a *dependency graph browser* to display the dependencies among the variables in the specification, the toolset includes an automated *consistency checker* to detect type errors, missing cases, circular definitions, and other types of application-independent errors, a *simulator* to allow users to symbolically execute the specification to ensure that it captures their intent, and an interface to a model checker called Spin [11, 12] that detects certain safety property violations.

An SCR requirements specification describes both the required system behavior and the system environment in terms of *monitored variables*, quantities that the system monitors, and *controlled variables*, quantities

that the system controls. To specify the required behavior concisely, SCR specifications use two types of auxiliary variables, *mode classes* and *terms*. Mode classes, whose values are called *system modes* (or simply *modes*), capture historical information, whereas terms have very general utility.

SCR requirements specifications contain a set of dictionaries and a set of tables. The dictionaries, which contain the static information in the specification, include a *variable dictionary*, which lists the name, data type, and initial value of each variable; the *type dictionary*, which provides the data type definitions; the *constant dictionary*, which defines the names and values of constants; the *specification assertion dictionary*, which contains statements of properties such as state invariants; and the *environmental assertion dictionary*, which describes constraints on the behavior of the monitored variables. For every variable other than a monitored variable, there is a corresponding *condition*, *event*, or *mode transition* table. Each table defines a mathematical function called a *table function*. For example, an event table describes the (post-transition) value of a controlled variable or term as a function of a mode and an event. The notation “ $\mathbf{QT}(c)$ ” denotes an *event*, defined as $\mathbf{QT}(c) = \neg c \wedge c'$, where the unprimed condition c is evaluated in the *current* state, and the primed condition c' is evaluated in the *new* state. Informally, “ $\mathbf{QT}(c)$ ” means that condition c becomes true.

To provide formal underpinnings for the SCR specifications, a formal model has been developed [9]. In the SCR model, the system is represented as a state machine that begins execution in some initial state and then responds to a sequence of input events, where an *input event* is an event that signals a change in some monitored variable. In particular, a system Σ is represented as a 4-tuple, $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is a function that maps an input event and the current state to a new state. The transform T is obtained from an SCR specification as the composition of the table functions. For T to be well-defined, the “direct” dependencies in the specification of a given variable in the new state on other variables in the new state must define a partial order. In SCR*, this partial order is verified to exist by the consistency checker, and is represented in the dependency graph browser as the *new state dependency graph*.

3 Recent Improvements to TAME

3.1 An Improved Induction Strategy

The induction strategy is the major TAME strategy for proving state invariants. This strategy sets up an induction proof for a state invariant by breaking the proof up into a base case and induction steps for the transitions, one for each kind of action. In the case where the invariant involves quantification over variables other than the automaton state, it is frequently the case that one wants to skolemize these variables in the inductive conclusion, and instantiate them in the inductive hypothesis with the resulting skolem constants. After some standard simplification on each branch, the induction strategy incorporates this skolemize-instantiate step on each induction branch of the proof. It then probes the branch to see whether a simple decomposition into cases followed by the application of the arithmetic, propositional logic, and other decision procedures of PVS will complete the proof of that branch—i.e., it checks to see if the branch is “trivial”. If so, the branch is proved; otherwise, the branch is unchanged. The induction strategy then returns the unproved branches to the user.

The induction strategy was an appropriate candidate for improvement using several of the PVS enhancements. Originally, an appropriate variant for each specific automaton required external compilation. Multiple versions were needed when there were state invariant lemmas both with and without quantification of non-state variables. Standard simplification of the branches expands certain definitions, including the automaton’s transition function. However, PVS’s proof rule EXPAND, normally used for this purpose, does not simply expand a definition, but does some additional steps that often, though not invariably, produce a desirable simplification. One case where there may not be a desirable simplification is when the expanded definition contains an IF-THEN-ELSE expression under a quantifier. In this case, the additional steps result in lifting the IF-THEN-ELSE to the top level, with the quantifier appearing in both the THEN and ELSE branches. When this happens in expanding the transition function in the inductive conclusion, performing the skolemize-instantiate step is difficult. Using REWRITE in place of EXPAND was one solution, but not completely satisfactory because REWRITE, too, comes with baggage, and a resulting loss in efficiency. In any case, automated support for the skolemize-instantiate step had to rely on knowing the exact formula numbers of the inductive conclusion and the inductive hypothesis. Finally, because the induction strategy only produces subgoals for the nontrivial proof branches, it was difficult to examine a saved proof and determine the correspondence of branches to cases. Knowing this correspondence is important in several contexts, including the case when one wants to go back and complete a partial proof. Previously, labeling the branches was best done interactively using PVS’s original, rather primitive, comment facility: incorporating a comment as an extra argument in an APPLY.

The improved induction strategy, which we call AUTO-INDUCT, avoids these problems. It need not be compiled; rather, it probes the body of the state invariant lemma being proved to retrieve data structures

from which the necessary proof steps can be computed. This information includes the list of actions, with their arguments, that correspond to the transitions of the subject automaton. The new induction strategy then computes the strategy that previously was compiled externally, and applies it. Part of the computed strategy is a call to another strategy used in simplifying the proof branches. At the point where this auxiliary strategy is called, there is enough information in the current proof goal to determine whether and how to do any coordinated skolemization-instantiation. The auxiliary strategy computes a sequence of steps to accomplish this, and then applies them.

The improvements in the induction strategy described above rely on the additional documentation and access functions for the PVS internals. Eliminating the remaining problems in the original induction strategy required several of the PVS enhancements. The problem with EXPAND was solved by using one of the new finer-grained steps that simply expands a definition, and does no more. The problem of knowing the locations of the inductive conclusion and inductive hypothesis was solved by using the new labeling capabilities in PVS: AUTO_INDUCT labels various parts of an induction goal as “inductive-conclusion”, “inductive-hypothesis”, “specific-precondition”, and so on. The auxiliary strategy called by AUTO_INDUCT can then skolemize “inductive-conclusion” and instantiate “inductive-hypothesis”. The new comment facility is used by AUTO_INDUCT to automatically label proof branches with their corresponding cases. An example proof goal with labels and comments is given in Section 4.

3.2 Other Improvements in the TAME Strategy Set

With the combination of tools to access and analyze PVS sequents, formulae, and expressions plus the enhancements to PVS, we have succeeded in implementing many of the improved or new TAME strategies discussed in [2]. The resulting TAME strategies are designed for simplicity of use. We list a sample below. We illustrate the usefulness of most of these examples in Section 4.

- The *invariant-lemma strategy* for applying previously proved state invariant lemmas in a proof formerly had two versions; it now has a single version called APPLY_INV_LEMMA. Its first argument is the name of a state invariant. It checks to see whether its second argument (if any) is a state, and if so, applies the corresponding state invariant lemma to that state. All remaining arguments except the state argument (if there is one) are used as instantiations to any top level universal quantifier in the invariant lemma. APPLY_INV_LEMMA also displays the invariant being applied as a comment.
- The *precondition-strategy* for introducing the explicit form of the precondition for the action of an inductive step into the hypotheses of the current proof goal is now called APPLY_SPECIFIC_PRECOND. It now provides two additional services. First, it displays this explicit form of the precondition as a comment in the proof. Second, if the precondition is a conjunction, APPLY_SPECIFIC_PRECOND separates the conjunct conditions into a list, and gives them additional labels of the form *specific-precondition-part-i*, where the index *i* indicates their original position in the conjunction.
- The strategy TIME_ETC_SIMP, intended to complete the proof in a branch when a human might say “it is now obvious”, has been replaced by the strategy TRY_SIMP. TRY_SIMP first hides formulae containing quantifiers, applies TIME_ETC_SIMP, and if the current subgoal is not proved, reveals the formulae it hid and applies TIME_ETC_SIMP again. Experimentation has shown TRY_SIMP to be frequently more efficient than TIME_ETC_SIMP, and never measurably less efficient.
- The strategy USE_EPSILON simplifies the application of the Hilbert ϵ -axiom. We have found USE_EPSILON useful in establishing certain properties of nondeterministic automata (e.g., hybrid automata in which there are tolerances in the amount of change in some state variables in a time-passage step; see [3]). Being able to analyze formulae and expressions in the proof goal with respect to content and type has enabled us to simplify the application of USE_EPSILON so that the user need no longer supply the domain type of the predicate to which the axiom is being applied, nor the full, sometimes complex, expression representing the predicate. Rather, the user need now only supply a hint as to which predicate to choose.
- The strategy DISCHARGE_HYPOTHESES is new. It removes hypotheses from any implications in the antecedent of a proof goal that can be deduced from the contents of the goal. While the standard PVS strategy ASSERT can sometimes be used to this purpose, it does not work if the hypotheses are of a certain complexity. DISCHARGE_HYPOTHESES can be used to avoid unnecessary case splits in the course of a proof.

Section 5 discusses a few new strategies useful in proving properties of SCR specifications.

3.3 Improvements in the Literacy of TAME Proofs

With the current set of TAME strategies, it is possible to maintain meaningful (sometimes multiple) labels on most if not all of the formulae in a proof goal. We have found labels to be useful not only in the support of more intelligent strategies, but in the course of an interactive proof.

Labels can serve as reminders of the source of a formula, and therefore aid in determining what point in reasoning has been reached in the proof, and in choosing the appropriate steps to take next. For example,

we have replaced PVS’s CASE rule with the TAME rule SUPPOSE. The only difference between the two rules is that SUPPOSE labels the introduced assumption *Suppose*, and in the companion proof branch in which the assumption must be proved, *Suppose not*. This documents the intended purposes of the formulae introduced.

Labels can also be used to increase proof portability. For example, the PVS proof rule INST (“instantiate”) frequently must specify the formula to instantiate, and previously, this could only be done by giving the formula number. We note that maintaining unique labels for individual formulae can be important for an analogous reason.

The current set of TAME strategies also makes liberal use of comments. As noted in Sections 3.1 and 3.2, our strategy AUTO_INDUCT labels proof branches with comments, and the strategies APPLY_INV_LEMMA and APPLY_SPECIFIC_PRECOND introduce comments that spell out the actual facts being introduced. Several other TAME strategies do this also. This information can be useful interactively in noting how the facts have been simplified. One place where it is particularly useful in proof scripts is in clarifying the effects of certain later proof steps—e.g., in determining exactly which general fact has been instantiated. Without also incorporating the sequent at the beginning of each proof branch as a comment, there is not yet enough information in TAME scripts to completely follow the reasoning in a proof. Exactly how much information to incorporate into a proof script is an open question, and is largely a matter of taste.

4 Examples

Figure 1 shows an example of how labels are used in induction case proof goals of TAME induction proofs. Note that the inductive conclusion has been “flattened” into three parts.

```

lemma_6_1.1 :
;;Case nu(timeof_action)
{-1,pre-state-reachable}
  reachable(prestate)
{-2,inductive-hypothesis}
  (((trains_part(basic(prestate))(r_theorem) = P)
    & ((gate_part(basic(prestate)) = fully_up) OR (gate_part(basic(prestate)) = going_up)))
    => first(prestate)(enterI(r_theorem)) > now(prestate) + gamma_down)
{-3,general-precondition}
  enabled_general(nu(timeof_action), prestate)
{-4,specific-precondition}
  enabled_specific(nu(timeof_action), prestate)
{-5,post-state-reachable}
  reachable(prestate WITH [now := now(prestate) + timeof_action])
{-6,inductive-conclusion_part_1,inductive-conclusion}
  (trains_part(basic(prestate))(r_theorem) = P)
{-7,inductive-conclusion_part_2,inductive-conclusion}
  ((gate_part(basic(prestate)) = fully_up) OR (gate_part(basic(prestate)) = going_up))
|-----
{1,inductive-conclusion_part_3,inductive-conclusion}
  first(prestate)(enterI(r_theorem)) > now(prestate) + timeof_action + gamma_down

```

Figure 1. Example of a labeled proof subgoal returned by AUTO_INDUCT.

The changes in the TAME proof of the subgoal in Figure 1 are a good example of the improvements now possible in TAME proof scripts. The proof of this subgoal now has only six steps where it previously had nine. The first step, an application of APPLY_SPECIFIC_PRECOND, results in an expansion and decomposition of the formula labeled *specific-precondition* into six individually labeled parts and the inclusion of a corresponding comment into the proof script. The next two steps are calls to APPLY_INV_LEMMA; previously one of these steps invoked the special version of the invariant-lemma strategy for universally quantified invariants. The next step, DISCHARGE_HYPOTHESES, now accomplishes simplifications that previously required three steps, two of which relied on formula numbers. Next, the step (INST “specific-precondition_part_5” “r_theorem”) replaces an INST step that required a formula number. The comment introduced by APPLY_SPECIFIC_PRECOND allows one to see from the proof script exactly which formula is instantiated by INST. Finally, the step TRY_SIMP now replaces two steps: a HIDE step that relies on a formula number, and a step TIME_ETC_SIMP.

5 Integrating TAME with SCR*

Currently, SCR specifications define deterministic automata, so their transitions can be represented by a function. Because this function is computed from many incremental updates in the state variables, proofs

that use the function representation for the transitions of an SCR machine are inefficient. In fact, dramatic increases in the speed of proofs have resulted from representing transitions by a relation rather than a function, in contrast with our experience with LV timed automata in [3]. Therefore, we adapted the TAME template, supporting theory **machine**, and induction strategy to accomodate this variation.

Translating SCR specifications into TAME specifications is straightforward. TAME specifications are simply PVS specifications with a special structure. Information in the type dictionary, constant dictionary, and variable dictionary of an SCR specification is translated into PVS type, constant, and variable declarations and the initial state predicate. The monitored and controlled variables, terms, and mode classes become the “basic” state variables (there are additional standard state variables reserved for encoding timing information). The “actions” of an SCR automaton are input events, which represent some change in a monitored quantity. Such actions have as a parameter the new value of the monitored quantity. The environmental assertion dictionary contains all information about constraints on how monitored quantities can change, and this information is translated into preconditions on the actions. The assertions in the specification assertion dictionary are translated into invariant lemmas in TAME format. Finally, the definition of the transition relation is obtained using the event, condition, and mode transition tables and the new state dependency graph, and is represented in terms of “update” functions for variables that are affected by a given action (input event). The update function for a variable is a translation of its table, based upon translations of predicates and expressions in the table that are indexed numerically in the natural way. This translation scheme has been automated for a significant subset of the SCR specification language. Figure 3 shows part of the automated translation of the event table for the state variable (“term”) **Overridden** shown in Figure 2, and how the corresponding update function for **Overridden** is used in the definition of the transition relation.

Experimentation with proving state and transition invariants for several example SCR specifications strongly suggests that there exists a uniform strategy that, for typical SCR applications, will suffice to prove many invariants automatically. We are continuing experiments aimed at the development of such a strategy.

Mode	Events	
High	False	@T(Pressure = TooLow OR Pressure = Permitted)
TooLow, Permitted	@T(Block=0n) WHEN Reset=0ff	@T(Pressure = High) OR @T(Reset=0n)
Overridden	True	False

Figure 2. Event table for Overridden.

```

statepred : TYPE = [ states -> bool ];
atT(P : statepred, s1,s2 : states):bool = P(s2) AND NOT P(s1);
Overridden_0_0(s1,s2 : states) : bool = FALSE;
Overridden_0_1(s1,s2 : states): bool =
  let e0 = (LAMBDA (s : states): Pressure(s) = TooLow OR Pressure(s) = Permitted) in atT(e0, s1, s2);
...
Overridden_assignment_0 (s1, s2 : states) : bool = TRUE;
Overridden_assignment_1 (s1, s2 : states) : bool = FALSE;
update_Overridden(s1,s2 : states) : bool =
  IF Pressure(s1) = High
    THEN IF Overridden_0_0(s1,s2) THEN Overridden_assignment_0(s1,s2)
        ELSIF Overridden_0_1(s1,s2) THEN Overridden_assignment_1(s1,s2)
        ELSE Overridden(s1)
    ENDIF
  ELSIF
    ...
  ELSE Overridden(s1)
  ENDIF;
trans(s_old : states, A : actions, s_new : states) : bool =
  s_new = CASES A OF Block(Block_value): s_old WITH [basic := basic(s_old) WITH
    [Block_part := Block_value,
     Overridden_part := update_Overridden(s_old,s_new),
     SafetyInjection_part := update_SafetyInjection(s_old,s_new)]];
...
ENDCASES;

```

Figure 3. Fragments of TAME translation of Overridden event table and the transition relation.

6 Discussion and Future Plans

Providing an interface such as TAME for a theorem proving system requires several capabilities within, or in relation to, that system. These include support for user-defined strategies or tactics and access to the data structures and some of the internal analysis tools used by the proof engine. TAME also relies on the term rewriting facilities in PVS. For some provers, it may be necessary to add capabilities: for PVS, we required additional low-level proof steps plus labeling and comment facilities.

Capabilities needed to provide what we call first-level interface support for a prover were noted in [21]. These include formal languages and parsers for the data structures used in the prover and its outputs, and a means to communicate changes of state between the prover and the interface. As indicated in the introduction, we plan to provide first-level interface support for TAME and to support the use of TAME through the SCR* toolset interface. To do so, we will need some additional capabilities along these lines.

With the capabilities we now have, we expect to improve our PVS interface further by providing strategies for PVS steps that we noted in [2] as potentially useful, e.g., in avoiding unnecessary case splitting. Examples are strategies for the skolemization and instantiation of embedded quantified formulae. The Lisp code needed in the strategies for analyzing formulae would almost certainly be simpler to create given access to analysis tools such as parsers already present in PVS.

Acknowledgments

We wish to thank our colleague Ralph Jeffords for helpful discussions about proof efficiency and SCR specifications, and our colleague Ramesh Bharadwaj for insightful comments on an earlier version of this paper. We also thank Natarajan Shankar and Sam Owre of SRI International, who implemented the enhancements to PVS that allowed us to improve TAME.

References

- [1] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*. IEEE Computer Society Press, 1996.
- [2] M. Archer and C. Heitmeyer. Human-style theorem proving using PVS. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 33–48. Springer-Verlag, 1997.
- [3] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.
- [4] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A Language for Specifying, Programming, and Validating Distributed Systems. Draft. MIT Laboratory for Computer Science, December, 1997.
- [5] J. Harrison. A Mizar mode for HOL. In *Proc. 9th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 203–220. Springer-Verlag, 1996.
- [6] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 109–122, Gaithersburg, MD, June 1995.
- [7] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.
- [8] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, Dec. 1994.
- [9] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.
- [10] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, Jan. 1980.
- [11] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [12] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997.
- [13] S. Kalvala. Annotations in formal specifications and proofs. *Formal Methods in System Design*, 5(1/2), 1994.
- [14] S. Kalvala. A formulation of TLA in Isabelle. In *Higher Order Logic Theorem Proving and Its Applications (HOL'95)*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 46–57. Springer-Verlag, 1995.
- [15] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
- [16] J. Rushby. Private communication. NRL, Jan. 1997.
- [17] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.
- [18] J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems, Lect. Notes in Comp. Sci. 863*. Springer-Verlag, 1994.
- [19] D. Syme. A new interface for HOL – ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications (HOL'95)*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 324–339. Springer-Verlag, 1995.
- [20] L. Théry. A proof development system for the HOL theorem prover. In *Higher Order Logic Theorem Proving and Its Applications (HUG'93)*, volume 780 of *Lect. Notes in Comp. Sci.*, pages 115–128. Springer-Verlag, 1993.
- [21] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. *Proc. Fifth ACM SIGSOFT Symp. on Software Development Environments, Software Engineering Notes*, 17(5), 1992.